METHOD AND SYSTEM FOR IDENTIFYING, FIXING, AND UPDATING SECURITY VULNERABILITIES

Technical Field

This invention relates to network communications for computers, and more particularly, to operating system security over open networks.

Background of the Invention

As Internet technology has advanced, users are able to access information on many different operating systems. Hackers take advantage of the open network architecture of the Internet, and attempt to gain access to operating systems without authorization. Hackers present a significant security risk to information stored on a operating system. In an effort to limit unauthorized access to operating system resources, many operating system communication security devices and techniques have been developed.

One security device and technique that has been used to secure operating system resources is an Internet scanner. A scanner enables a user to find and report security vulnerabilities in network-enabled operating systems. The scanner can run a list of checks, or exploits, that verify the presence or absence of known security vulnerabilities. The exploits' findings are displayed to the user, and reports may be generated showing the discovered security vulnerabilities and methods for fixing them.

Although scanners are very useful, they lack services that users need to adequately protect their operating systems. The release cycle of a scanner is long compared to the time required to develop and test individual security checks. New security vulnerabilities are introduced very rapidly, and must be found and addressed in real-time. Because hackers create problems in systems on a minute-to-minute basis, a scanner must be updated constantly to be most valuable to a user.

What is needed is a method and system for providing updated exploit information in a short time period. A scanner needs to have its components sufficiently separated so that individual information used in the scanner can be updated independently. A scanner's individual security exploits need to be updated and released independently of the entire scanner's release cycle. The exploit information needs to be available on an per-exploit basis so that minor, but important,

modifications can be made without affecting the entire system. In addition, exploit information, including help information, needs to be updated independently of the exploit itself.

A further need in the art exists for a user-friendly scanner with the above update capability. The user needs to be able to use the system without needing to know whether the exploits are included in the scanner or are separately installed via update procedures.

A further need in the art exists for a scanner with the above update capability that includes mutual authentication procedures. Constant update packages necessitate ensuring that the scanner will only load legitimate updates, and that updates will only be loaded into legitimate scanners.

Summary of the Invention

The present invention satisfies the above-described needs by providing a system and method for identifying, fixing, and updating security vulnerabilities in host computers. In an exemplary embodiment, the identifying, fixing, and updating capabilities can be done by communication between a scanner with plug-in capability, an operating system, and an express update package.

The express update package can contain exploit plug-in modules, resource plug-in modules, dat files, and help files. The exploit plug-in modules and the resource plug-in modules can be dynamic-link libraries (DLLs). The exploit plug-in module can contain exploit objects and the resource plug-in module can contain resource objects. The exploit objects can contain exploits and the resource objects can contain resources. An exploit can be an individual security check that is done on a computer or systems. A resource can be an individual resource that is used by the scanner, and can include data, executable code, or a network connection.

The present invention can yield an architectural solution that allows the exploits within the scanner, and the exploits in the express update package, to function with no knowledge of each other. Because the exploit objects and the resource objects can hide their implementation details behind standard interfaces, they may be managed and manipulated by the scanner without knowledge of their internal make-up. This architectural solution can allow existing exploits to be modified and incorporated into the scanner without updating the entire scanner. In addition, new exploits may be added to the scanner without updating the entire scanner. The present invention can be user-friendly in that the user needs no knowledge regarding

whether the exploits are included in the scanner's installation package or are separately installed via update procedures.

The present invention can also include mutual authentication procedures. The authentication procedures can enable the scanner to load only legitimate plug-in modules, and can provide that plug-in modules can only be loaded into legitimate scanners.

Brief Description of the Drawings

FIG. 1 is a block diagram of a personal computer that provides an exemplary operating environment for an exemplary embodiment of the present invention.

FIG. 2 is a block diagram illustrating internal program objects of an exemplary embodiment which can report actions between an operating system, a scanner with plug-in capability, and an express update package.

FIG. 3 is a screen display of information the user can access in an Inventory folder in an exemplary embodiment of the present invention.

FIG. 4 is a screen display of information the user can access in a Vulnerabilities folder in an exemplary embodiment of the present invention.

FIG. 5 is a screen display of information the user can access in a Services folder in an exemplary embodiment of the present invention.

FIG. 6 is a screen display of information the user can access in a Accounts folder in an exemplary embodiment of the present invention.

FIG. 7 is a screen display showing the different views the user can access in an exemplary embodiment of the present invention.

FIG. 8 is a flowchart diagram illustrating an exemplary method for identifying, fixing, and updating security vulnerabilities in a host computer or computers.

FIG. 9 is a flowchart diagram illustrating an exemplary method for initializing a scanner.

FIG. 10 is a flowchart diagram illustrating an exemplary method for running load security and loading a plug-in module.

FIG. 11 is a flowchart diagram illustrating an exemplary method for initializing a policy manager.

FIG. 12 is a flowchart diagram illustrating an exemplary method for running activation security and creating available exploit/ resource objects.

FIG. 13 is a flowchart diagram illustrating an exemplary method for getting license, policy and host information.

FIG. 14 is a flowchart diagram illustrating an exemplary method for getting policy information.

FIG. 15 is a flowchart diagram illustrating an exemplary method for running exploits.

FIG. 16 is a flowchart diagram illustrating an exemplary method for running built-in exploits.

FIG. 17 is a flowchart diagram illustrating an exemplary method for running plug-in exploits.

FIG. 18 is a flowchart diagram illustrating an exemplary method for determining an optimal order for running plug-in exploits.

Detailed Description of Exemplary Embodiments

The present invention can be a method of identifying, fixing, and updating security vulnerabilities in a host computer or computers. In an exemplary embodiment, the identifying, fixing, and updating capabilities can be done by communication between a scanner with plug-in capability, an operating system, and a plug-in module. Because the exploit objects and the resource objects can hide their implementation details behind standard interfaces, they may be managed and manipulated by the scanner without knowledge of their internal make-up. This architectural solution can allow existing exploits to be modified and incorporated into the scanner without updating the entire scanner. In addition, new exploits may be added to the scanner without updating the entire scanner. Mutual authentication procedures can also be used to ensure that only legitimate scanners and express update package contents are used.

FIG. 1 is a block diagram of a personal computer that provides an exemplary operating environment. FIG. 2 is a block diagram illustrating internal program objects. FIGS. 3 - 7 are screen displays showing the user interface (UI) component for an exemplary embodiment of the present invention. FIGS. 8 - 18 are flowchart diagrams illustrating exemplary methods for identifying, fixing, and updating security vulnerabilities in a host computer or computers.

Although the preferred embodiment will be generally described in the context of a program and an operating system running on a personal computer, those skilled in the art will recognize that the present invention also can be implemented in

conjunction with other program modules for other types of computers. Furthermore, those skilled in the art will recognize that the present invention may be implemented in a stand-alone or in a distributed computing environment. In a distributed computing environment, program modules may be physically located in different local and remote memory storage devices. Execution of the program modules may occur locally in a stand-alone manner or remotely in a client/server manner. Examples of such distributed computing environments include local area networks of an office, enterprise-wide computer networks, and the global Internet.

The detailed description which follows is represented largely in terms of processes and symbolic representations of operations by conventional computer components, including a central processing unit (CPU), memory storage devices for the CPU, display devices, and input devices. Furthermore, these processes and operations may utilize conventional computer components in a heterogeneous distributed computing environment, including remote file servers, remote compute servers, and remote memory storage devices. Each of these conventional distributed computing components is accessible by the CPU via a communications network.

The processes and operations performed by the computer include the manipulation of signals by a CPU or remote server and the maintenance of these signals within data structures resident in one or more of the local or remote memory storage devices. Such data structures impose a physical organization upon the collection of data stored within a memory storage device and represent specific electrical or magnetic elements. These symbolic representations are the means used by those skilled in the art of computer programming and computer construction to most effectively convey teachings and discoveries to others skilled in the art.

For the purposes of this discussion, a process is generally conceived to be a sequence of computer-executed steps leading to a desired result. These steps generally require physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical, magnetic, or optical signals capable of being stored, transferred, combined, compared, or otherwise manipulated. It is conventional for those skilled in the art to refer to these signals as bits, bytes, words, data, objects, properties, flags, types, identifiers, values, elements, symbols, characters, terms, numbers, points, records, images, files or the like. It should be kept in mind, however, that these and similar terms should be associated with appropriate physical quantities for computer operations, and that these terms are merely

conventional labels applied to physical quantities that exist within and during operation of the computer.

It should also be understood that manipulations within the computer are often referred to in terms such as comparing, selecting, viewing, getting, giving, etc. which are often associated with manual operations performed by a human operator. The operations described herein are machine operations performed in conjunction with various input provided by a human operator or user that interacts with the computer.

In addition, it should be understood that the programs, processes, methods, etc. described herein are not related or limited to any particular computer or apparatus, nor are they related or limited to any particular communication network architecture. Rather, various types of general purpose machines may be used with program modules constructed in accordance with the teachings described herein. Similarly, it may prove advantageous to construct a specialized apparatus to perform the method steps described herein by way of dedicated computer systems in a specific network architecture with hardwired logic or programs stored in nonvolatile memory, such as read only memory.

Referring now to the drawings, in which like numerals represent like elements throughout the several figures, aspects of the present invention and the preferred operating environment will be described.

The Operating Environment

FIG. 1 illustrates various aspects of an exemplary computing environment in which the present invention is designed to operate. Those skilled in the art will immediately appreciate that FIG. 1 and the associated discussion are intended to provide a brief, general description of the preferred computer hardware and program modules, and that additional information is readily available in the appropriate programming manuals, user's guides, and similar publications.

The Computer Hardware

FIG. 1 illustrates a conventional personal computer **10** suitable for supporting the operation of the preferred embodiment of the present invention. As shown in FIG. 1, the personal computer **10** operates in a networked environment with logical connections to a remote computer **11**. The logical connections between the personal computer **10** and the remote computer **11** are represented by a local area network **12** and a wide area network **13**. Those of ordinary skill in the art will recognize that in

this client/server configuration, the remote computer **11** may function as a file server or computer server.

The personal computer **10** includes a CPU **14**. The personal computer also includes system memory **15** (including read only memory (ROM) **16** and random access memory (RAM) **17**), which is connected to the CPU **14** by a system bus **18**. The preferred computer **10** utilizes a BIOS **19**, which is stored in ROM **16**. Those skilled in the art will recognize that the BIOS **19** is a set of basic routines that helps to transfer information between elements within the personal computer **10**. Those skilled in the art will also appreciate that the present invention may be implemented on computers having other architectures, such as computers that do not use a BIOS, and those that utilize other microprocessors, such as the "MIPS" or "POWER PC" families of microprocessors from Silicon Graphics and Motorola, respectively.

Within the personal computer **10**, a local hard disk drive **20** is connected to the system bus **18** via a hard disk drive interface **21**. A floppy disk drive **22**, which is used to read or write a floppy disk **23**, is connected to the system bus **18** via a floppy disk drive interface **24**. A CD-ROM or DVD drive **25**, which is used to read a CD-ROM or DVD disk **26**, is connected to the system bus **18** via a CD-ROM or DVD interface **27**. A user enters commands and information into the personal computer **10** by using input devices, such as a keyboard **28** and/or pointing device, such as a mouse **29**, which are connected to the system bus **18** via a serial port interface **30**. Other types of pointing devices (not shown in FIG. 1) include track pads, track balls, pens, head trackers, data gloves and other devices suitable for positioning a cursor on a computer monitor **31**. The monitor **31** or other kind of display device is connected to the system bus **18** via a video adapter **32**.

The remote computer **11** in this networked environment is connected to a remote memory storage device **33**. This remote memory storage device **33** is typically a large capacity device such as a hard disk drive, CD-ROM or DVD drive, magneto-optical drive or the like. The personal computer **10** is connected to the remote computer **11** by a network interface **34**, which is used to communicate over the local area network **12**.

As shown in FIG. 1, the personal computer **10** is also connected to the remote computer **11** by a modem **35**, which is used to communicate over the wide area network **13**, such as the Internet. The modem **35** is connected to the system bus **18** via the serial port interface **30**. The modem **35** also can be connected to the public switched telephone network (PSTN) or community antenna television (CATV)

network. Although illustrated in FIG. 1 as external to the personal computer **10**, those of ordinary skill in the art will quickly recognize that the modem **35** may also be internal to the personal computer **11**, thus communicating directly via the system bus **18**. It is important to note that connection to the remote computer **11** via both the local area network **12** and the wide area network **13** is not required, but merely illustrates alternative methods of providing a communication path between the personal computer **10** and the remote computer **11**.

Although other internal components of the personal computer **10** are not shown, those of ordinary skill in the art will appreciate that such components and the interconnection between them are well known. Accordingly, additional details concerning the internal construction of the personal computer **10** need not be disclosed in connection with the present invention.

Those skilled in the art will understand that program modules such as an operating system **36** and data are provided to the personal computer **10** via computer-readable media. In the preferred computer, the computer-readable media include the local or remote memory storage devices, which may include the local hard disk drive **20**, floppy disk **23**, CD-ROM or DVD **26**, RAM **17**, ROM **16**, and the remote memory storage device **33**. In the preferred personal computer **10**, the local hard disk drive **20** is used to store data and programs, including the operating system **36** and the scanner **37**.

The focus of the express update package **38** is described below in a manner that relates to its use in a scanner **37** with plug-in capability of FIG. 1. This description is intended in all respects to be illustrative rather than restrictive. Alternative embodiments will be apparent to those skilled in the art.

The Internal Objects

FIG. 2 is a block diagram illustrating internal program objects of an exemplary embodiment which can report actions between an operating system **36**, a scanner **37** with plug-in capability, and an express update package **38**. The scanner **37** can include a UI **205**, a session manager **235**, a thread manager **260**, host-scanning threads **265**, an engine **270**, an exploit manager **230**, and a resource manager **220**. The present invention also can access a registry **285**, a database **290**, and some scanner log files **295**.

An express update package **38** can contain exploit plug-in modules **299**, resource plug-in modules **297**, dat files **293**, and help files **292**. The exploit plug-in

modules **299** and the resource plug-in modules **297** can be DLLs. An exploit plug-in module **299** can contain one or more exploit objects **294**. An exploit object **294** can be a container for a plug-in exploit **291**. The plug-in exploit **291** can be an individual exploit, or security check, that is done on the host computer or computers.

The resource plug-in module **297** can contain one or more resource objects **298**. A resource object **298** can contain a plug-in resource **289**. The plug-in resource **289** can be an individual resource that is used by the scanner **37**. The resources may include data, executable code, or a network connection. Examples of resources are a list of known accounts on a host or an open file transfer protocol (FTP) connection. Because the plug-in exploits **291** that produce and consume shared resources can be added to the scanner **37** installation dynamically, the resources can also have plug-in capability, and can be packaged and delivered separately from the scanner **37**. Like the exploit objects **294**, the resource objects **298** can expose standard interfaces enabling the scanner **37** to manage them without any knowledge of their function or purpose.

There can be several basic types of resources. A mandatory resource can be one that an exploit must have in order to perform successfully. An optional resource can be one that an exploit can use if the resource exists, but is not necessary for the exploit to function properly. A create-on-demand (C-O-D) resource can be a resource that is created the first time it is requested. Afterwards, when a requested C-O-D resource already exits, the requestor will get the C-O-D resource, instead of having the resource recreated. A create-unique resource can be a resource that is always created afresh when requested, so that the requestor is guaranteed that the resource is unique and will not be accessed or used by any other requestor.

When a resource is created, it can be assigned a namespace based on its scope. The scope of a resource can be an indication of its specificity, and can include host-specific resources, session-specific resources, and global resources. A host-specific resource can only be used by exploits running against the host and in the session to which a resource belongs. A session-specific resource can be used only by exploits running against a host in the scan session to which the resource belongs. A global resource can be used by any exploit in any scan session.

The separation of the resources and the exploits in the resource objects **298** and the exploit objects **294** is a key idea in the architecture, and has substantial benefits. The scanner **37** can be more efficient because a resource required by multiple exploits only needs to be created once, instead of once for each of the

exploits. The scanner **37** can also be more flexible because the resources and the exploits may be updated independently of each other. In addition, because the exploit objects **294** and the resource objects **298** can hide their implementation details behind standard interfaces, they may be managed and manipulated by an application, such as the scanner **37**, that does not know of their internal make-up. This can yield an architectural solution where the exploit need have no knowledge of the other exploits that produce or consume its resources.

The dat file **293** can store exploit attribute information for all of the exploits in the express update package **38**. There can be one dat file **293** for each exploit plug-in module **299**. When queried for its attribute information, the exploit reads the pertinent data from the dat file **293**. The separation of the exploit attribute information from the exploit object **294** itself allows the exploit object **294** and the dat file **293** to be updated independently. This separation also allows only some dat files **293** to be updated if other dat files **293** do not need to be updated or changed.

The help file **292** can contain on-line help information associated with the exploit objects **294** that can be contained in the exploit plug-in module **299**. When a user requests help, the scanner **37** can read the information from the file and display it in the UI **205**. The separation of the exploit's help information from the exploit object **294** can allow the help file **292** and the exploit object **294** to be updated independently of one another.

An exploit manager **230** manages the exploit objects **294** and a resource manager **220** manages the resource objects **298**. The exploit manager **230** and the resource manager **220** can access the exploit objects **294** contained in the exploit plug-in module **299** and the resource objects **298** contained in the resource plug-in module **297**. The exploit manager **230** and the resource manager **220** can convey exploit objects **294** and resource objects **298** to the policy manager **215** and plug-in engine **275**.

The UI **205** can exchange information with the user. The UI **205** can include a policy editor **210** and a policy manager **215**. The policy editor **210** can allow a user to examine, modify, create and configure policies; acquire on-line documentation about any exploit; perform keyword searches on the on-line documentation; and alter the current presentation of information based on category choices or search results. Policy information can be a scan configuration, consisting of a set of enabled exploits and any necessary parameters for those exploits. The policy manager **215** can pass exploit policy information to and from the policy editor **210** via a scanpolicy object

245. In addition, the policy manager **215** can acquire exploit objects **294** from the exploit manager **230** and resource objects **298** from the resource manager **220**, and then can create the scanpolicy objects **245**. A scanpolicy object **245** can be a container for policy information, and can expose interfaces enabling the scanner **37** components to query it for this information. When used in a scan session, the scanpolicy object **245** can be stored in a session object **240**.

The session object **240** can contain all information necessary to run a scan session. A scan session can run a series of exploits for one or more hosts. The engine **270** can query the session object **240**. The session object **240** can contain: the scanpolicy object **245** identifying enabled exploits and their parameters; a list of hosts to scan; a master exploit list **250**; a master resource list **255**, and a license file. The session object **240** can construct the master exploit list **250** and the master resource list **255**. The scanpolicy object **245** can be built by the policy manager **215** and the scanpolicy object **245** can be used to construct the master exploit list **250** and master resource list **255**. The master exploit list **250** can contain information about all the plug-in exploits **291** enabled for a scan session. For each plug-in exploit **291**, the master exploit list **250** can contain its exploit object **294** and information about any resource objects **298** produced or consumed by the exploit. The master resource list **255** can contain information about the resources needed for a scan session. For each resource, the list contains its resource object **298**, and information about any exploits that produce or consume the resource.

A session manager **235** can contain and manage the scan sessions as represented by session objects **240**. The session manager **235** can exchange scan configuration setting information with a thread manager **260**. The session manager **235** can ensure that host-scanning threads **265** are allocated equitably among the various session objects **240**. The UI **205** and the engine **270** can query the session manager **235** for information on what host to scan, scan configuration settings, etc. The thread manager **260** can get information from the session manager **235** that describes when a new session has been created, the number of hosts in a session and scan configuration parameters. The thread manager **260** can also create the host-scanning threads **265**. The host-scanning thread **265** can tell the thread manager **260** when it is finished with a scan session. Otherwise, the host-scanning thread **265** can communicate with the session manager **235** to get host information.

The engine **270** can run the built-in exploits. The plug-in engine **275** can be included in the engine **270** and can run the plug-in exploits **291**, and can contain the

target object **280** and a copy of the master exploit list **250** and master resource list **255**. The resources that are produced and consumed by various exploits can create dependencies among exploits. The plug-in engine **275** can run the plug-in exploits **291** in a particular order. There can be a plug-in engine **275** instance for each host. The plug-in engine **275** can query the session manager **235** and can make its own copies of the master exploit list **250** and the master resource list **255**. The plug-in engine **275** can then use its copy of the master exploit list **250** and the master resource list **255** to run the plug-in exploits **291**. The master exploit list **250** and the master resource list **255** can have information on each exploit and resource. In particular, these lists **250** and **255** can be used to determine the order of running the plug-in exploits **291**.

The target objects **280** can be containers that provide a means of communication between the plug-in engine **275** and the exploit objects **294**. The plug-in engine **275** can create a target object **280**, and can reuse the same target object **280** for each plug-in exploit **291** run against a host. Before executing the plug-in exploit, the plug-in engine **275** can query the exploit object **294** for information on the resources it requires. The plug-in engine **275** can acquire the required resource objects **298** from the resource manager **220** and can put them into the target object **280**. The target object **280** can pass the required resource objects **298** into the exploit object **294**. The plug-in exploit **291** can then be run, and the exploit object **294** can pass the scan result information into the target object **280**. The target object **280** can then return this scan result information to the plug-in engine **275**, which updates the UI **205**, the database **290**, and the scanner log file **295**.

The Screen Displays

Turning now to FIGS. 3-7, screen displays showing the UI **205** component for an exemplary embodiment of the present invention are shown.

FIG. 3 is a screen display of the detailed information in an Inventory folder the user can access in an exemplary embodiment of the present invention. Under an Inventory folder **305**, there can be a FlexChecks folder **320** and a Common Settings folder **310**. The FlexChecks folder **320** can enable a user to write his own checks or exploits to plug in the scanner **37**. The Common Settings folder **310** can contain global settings that can be enabled for a policy. These settings may apply to multiple vulnerability checks. An example of information found in the Common Settings folder **310** is the HTTP Ports folder **315**. This folder can have two options for using

the HTTP Ports setting. One option, the HTTP Ports option **325** allows the user to default and look for port 80 or port 8080, in addition to looking for the specified port. The HTTP Secure Ports option **330** can only look for the specified port or ports.

FIG. 4 is a screen display of the detailed information the user can access in a Vulnerabilities folder in an exemplary embodiment of the present invention. The Vulnerabilities folder **405** can contain and describe the exploits that check the holes in the operating system **36** which could allow an intruder to gain information and allow improper access. Under the Vulnerability folder **405**, there can be a Denial-of-Service folder **410** and a Standard folder **425**. The Denial-of-Service folder **410** lists the exploits that have the potential of shutting down a system or service. These include the E-mail **415** and Instant Messaging **420** categories.

The Standard folder **425** can hold numerous categories of standard vulnerabilities, including Backdoors, Browser, E-mail, and Firewalls categories. The exploit details that can be shown in the Vulnerabilities folder **405** include: risk levels, attack names, platforms, descriptions, remedies, references, common vulnerabilities and exposures (CVE), and links to other sources.

FIG. 5 is a screen display of information a user can access in a Services folder in an exemplary embodiment of the present invention. The Services folder **505** can list the types of services that the scanner **37** will attempt to connect to on the user's network. The Service folder **505** can include information such as transmission control protocol (TCP) Services **510**, which can attempt to connect to all well-known TCP-based service ports.

FIG. 6 is a screen display of information a user can access in an Accounts folder in an exemplary embodiment of the present invention. The Accounts folder **605** can list the types of accounts the scanner **37** checks as it scans the user's network. When accessing the Accounts folder **605**, the UI **205** can access an list of accounts under a particular folder's name. For example, a NetBIOS folder **610** can check for accounts with NetBIOS names that can be used to identify a computer. A NetBIOS is an application programming interface (API) that can be used by application programs on a local area network.

FIG. 7 is a screen display showing the different views the user can access in an exemplary embodiment of the present invention. A Standard View **705** can be the default view that displays the exploits in a normal order. A Module View **710** can display which exploits are contained in which plug-in modules. A Risk View **715** can display the exploits in folders according to their level of risk: high, medium, or low.

A Category View **720** can display the exploits according to what category of vulnerability it falls under. Some examples of the Category View **720** are E-mail vulnerabilities or Backdoor vulnerabilities. A Built-in/Plug-in View **725** allows the user to see which exploits are built-in exploits and plug-in exploits **291**. Outside of this view in the policy editor **210**, the user cannot distinguish between the plug-in exploits **291** and built-in exploits. The user cannot distinguish this difference because each built-in exploit has a dummy plug-in object. The dummy plug-in object can be stored in the exploit plug-in module **299**. The dummy plug-in objects can look like the regular plug-in exploit objects **294** but they are never executed. The UI **205**, the policy manager **215**, and the help file **292** can access the dummy plug-in object.

The Plug-in Capability

FIG. 8 is flowchart diagram illustrating an exemplary method for identifying, fixing, and updating security vulnerabilities in a host computer or computers. In routine **805**, the scanner **37** can initialize. In routine **810**, the UI **205** can get the license, policy, and host information. In step **811**, the policy manager **215** can create the scanpolicy object **245**. In routine **812**, the policy editor **210** can allow the policy information to be edited. The policy information includes which exploits are enabled, configuration parameters, and common-setting resources. In step **815**, the user can initiate a scan. In step **820**, the UI **205** can create the session object **240** and the session object **240** can use the scanpolicy object **245** to create the master exploit list **250** and the master resource list **255**. In step **825**, the UI **205** can register the session object **240** with the session manager **235**. In step **830**, the thread manager **260** can start host-scanning threads **265** for the scan session. In step **835**, the first host-scanning thread can ask the session manager **235** for host, license, and policy information. In routine **840**, the engine **270** can run the exploits. In step **845**, the engine **270** can repeat steps **835-840** for the remaining hosts in the scan session.

Initializing a Scanner

FIG. 9 is a flowchart diagram illustrating an exemplary routine **805** for initializing a scanner **37** as set forth in FIG. 8. In step **905**, the exploit manager **230** and resource manager **220** can enumerate the installed exploit plug-in modules **299**, resource plug-in modules **297**, exploit objects **294**, and resource objects **298**. In routine **910**, load security can be run for each exploit plug-in module **299** and

resource plug-in module **297**, and these plug-in modules **299** and **297** can be loaded in the scanner **37**. In routine **915**, the policy manager **215** can initialize.

FIG. 10 is a flowchart diagram illustrating an exemplary routine **910** for running load security and loading the plug-in modules **299** and **297** in the scanner **37** as set forth in FIG. 9. In step **1005**, the scanner **37** and plug-in modules **299** and **297** can be digitally signed prior to release. In step **1010**, the exploit manager **230** can run load security on the exploit plug-in modules **299** and the resource manager **220** can run load security on the resource plug-in modules **297**. The purpose of load security can be to ensure that only legitimate applications (such as a legitimate scanner **37**) may load the plug-in modules **299** and **297** and that the scanner **37** only loads legitimate plug-in modules **299** and **297**. Load security can verify the digital signature of the plug-in modules **299** and **297**. In step **1011**, if the digital signature is incorrect, the load security is unsuccessful, and the scanner **37** will not load the plug-in modules **299** and **297**. In step **1015**, if the digital signature is correct, then the load security is successful, and the exploit manager **230** can load the exploit plug-in module **299** into the scanner **37**, and the resource manager **220** can load the resource plug-in module **297** into the scanner **37**. In step **1020**, the plug-in module **299** or **297** can run load security, and can verify the digital signature of the scanner **37**. In step **1021**, if load security is unsuccessful, the plug-in module **299** or **297** can remove itself from the scanner **37**. In step **1025**, if load security is successful, the exploit plug-in module **299** can allow the exploit manager **230** to access its internal functions, or the resource plug-in module **297** can allow the resource manager **220** to access its internal functions.

FIG. 11 is a flowchart diagram illustrating an exemplary routine **915** for initializing the policy manager **215** as set forth in FIG. 9. In step **1105**, the policy manager **215** can ask the exploit manager **230** and resource manager **220** what exploits and resources are available. In step **1110**, the exploit manager **230** and resource manager **220** can go to the registry **285** to find out what exploits and resources are available. In step **1115**, the exploit manager **230** and the resource manager **220** can create maps indicating which of the plug-in modules **299** or **297** contain the available exploit objects **294** and the available resource objects **298**. In step **1120**, the policy manager **215** can ask the exploit manager **230** and the resource manager **220** to get all the exploits objects **294** and common-setting resource objects **298**. The common-setting resource objects **298** can contain configuration information that can be used by multiple exploits. In routine **1125**, activation security can be run

to ensure that the scanner **37** and the exploit objects **294** and resource objects **298** are legitimate, and the available exploit objects **294** and common-setting resource objects **298** can be created. In step **1130**, the exploit manager **230** can get the exploit objects **294** and the resource manager **220** can get the resource objects **298**. The exploit manager **230** can return the exploit objects **294** and the resource manager **220** can return the resource objects **298** to the policy manager **215**. The policy manager **215** can then query the exploit objects **294** and resource objects **298** for exploit attribute and resource configuration information.

FIG. 12 is a flowchart diagram illustrating an exemplary routine **1125** for running activation security and creating the available exploit objects **294** and resource objects **298** as set forth in FIG. 11. Activation security can implement a modified SKID3 protocol exchange where both sides demonstrate their knowledge of a shared secret. In step **1205**, the exploit manager **230** and the resource manager **220** can find out if the exploit plug-in module **299** or the resource plug-in module **297** know the shared secret. In step **1206**, if the exploit plug-in module **299** or the resource plug-in module **297** do not know the shared secret, the exploit manager **230** or resource manager **220** can refuse to create the exploit object **294** or resource object **298**. In step **1210**, once the plug-in module **299** or **297** has demonstrated its knowledge of the shared secret, the exploit manager **230** or resource manager **220** can demonstrate its knowledge of the shared secret. In step **1211**, if the exploit manager **230** or the resource manager **220** do not know the shared secret, the plug-in module **299** or **297** can refuse access to its class factory. The class factory can be used to build the instances of the exploit object **294** or resource object **298**. In step **1215**, if the exploit manager **230** or the resource manager **220** has demonstrated its knowledge of the shared secret, the exploit manager **230** or resource manager **220** can allow access to the class factory to create the exploit object **294** or resource object **298** contained in the plug-in module **299** or **297**.

Getting License, Policy and Host Information

FIG. 13 is a flowchart diagram illustrating an exemplary routine **810** for getting license, policy and host information as set forth in FIG. 8. In step **1305**, the UI **205** can specify the license information. In routine **1310**, the UI **205** can specify the policy information. In step **1315**, the UI **205** can specify the host information.

FIG. 14 is a flowchart diagram illustrating an exemplary routine **812** for getting policy information as set forth in FIG. 8. In step **1410**, the policy editor **210**

can allow the user to examine, modify and configure the policy settings of the available exploits and resources. In step **1415**, the policy editor **210** can store the choices in a policy file.

Running Exploits

FIG. 15 is a flowchart diagram illustrating an exemplary routine **840** for running the exploits as set forth in FIG. 8. The engine **270** includes a plug-in engine **275**. The exploits can be denial-of-service (DoS) exploits or standard exploits. A DoS exploit may initiate a condition on a scanned host that damages its ability to perform some needed function. Typical DoS exploits may crash a running service. More severe DoS exploits may crash the host itself. DoS exploits, if enabled, can be scheduled to execute last to avoid interfering with the scanner's **37** ability to successfully execute other exploits. The standard exploits can be exploits that do not initiate a DoS condition. In routine **1505**, the engine **270** can run the standard built-in exploits. In routine **1510**, the plug-in engine **275** can run the standard plug-in exploits 291. In routine **1515**, the plug-in engine **275** can run the DoS plug-in exploits **291**. In routine **1520**, the engine **270** can run the DoS built-in exploits.

FIG. 16 is a flowchart diagram illustrating an exemplary routine **1505** or **1520** for running the built-in exploits as set forth in FIG. 15. The built-in exploits are in a list and the engine **270** can attempt to run the built-in exploits in the order they are put in the list. However, the engine **270** cannot run a built-in exploit if its resources are not yet available. Other exploits may need to be run to create the resources needed to run a particular built-in exploit. Multiple passes through the list may thus be necessary for the engine **270** to run all the built-in exploits in the list. In step **1605**, the engine **270** can attempt to run the exploit at the top of the built-in exploit list. In step **1610**, the engine **270** can record the scan result information to the database **290** and the scanner log file **295** and sends the scan result information to the UI **205** to display. In step **1615**, the engine **270** can repeat steps **1605-1610** for the remaining built-in exploits in the list.

FIG. 17 is a flowchart diagram illustrating an exemplary routine **1510** or **1515** for running the plug-in exploits **291** as set forth in FIG. 15. In step **1705**, the session object **240** can use the scanpolicy object **245** to create the master exploit list **250** and the master resource list **255**. In step **1710**, the plug-in engine **275** can make copies of the master exploit list **250** and the master resource list **255**. In step **1715**, the plug-in engine **275** can get the host information and resources for the first exploit. In step

1720, the plug-in engine 275 can create a target object 280. In step 1721, the plug-in engine 275 can put the host information and resources in the target object 280. In step 1725, the plug-in engine 275 can pass the target object 280 to the exploit object 294. In step 1730, the plug-in engine 275 can run the plug-in exploit 291. In step 1735, the exploit object 294 can add the log and scan result information to the target object 280. In step 1740, the exploit object 294 can pass the target object 280 back to the plug-in engine 275. In step 1745, the plug-in engine 275 can query the target object 280 for the log and scan result information. In step 1750, the plug-in engine 275 can record the scan result information to the database 290 and the scanner log file 295 and sends it to the UI 205 for display. In step 1755, the plug-in engine 275 can get the host and resource information for the next exploit and can repeat steps 1721-1750.

FIG. 18 is a flowchart diagram illustrating an exemplary method for determining the optimal order for running both the standard and the DoS plug-in exploits 291. The existence of shared resources that are produced and consumed by various exploits can imply possible dependencies among exploits. These possible dependencies can exist only for mandatory resources. The plug-in engine 275 can schedule all producers of a mandatory shared resource to execute before any of the consumers of that resource. Each plug-in engine 275 can make a copy of the master exploit list 250 and master resource list 255 for each scanned host because the copied master lists 250 and 255 can continually change during each host scan.

The master exploit list 250 can be divided into four sections. The first section can include the exploits that neither produce nor consume resources. In step 1805, the plug-in engine 275 can first run these plug-in exploits 291.

The second section of the master exploit list 250 can include the exploits that only produce resources. In step 1810, the plug-in engine 275 can run these plug-in exploits 291. After each exploit is run, the copied master resource list 255 can be updated to indicate which resources have been created.

The third section of the master exploit list 250 can include the exploits that both produce and consume. In step 1815, the plug-in engine 275 can run these plug-in exploits 291. The plug-in engine 275 can ask each of these exploits which resources the exploit needs to run. For example, the copied master exploit list 250 can indicate that exploit 1 needs resources A, B, and C to run. The plug-in engine 275 can then go to the copied master resource list 255 and find out that exploits 5, 8, and 10 need to run to produce resources A, B, and C. Exploits 5, 8 and 10 can be run, producing A, B, and C. Then exploit 1 can be run using A, B, and C. This procedure

of scheduling exploits that produce required resources to run prior to consumers of those resources can apply to exploits 5, 8, and 10. To make the process run smoothly, cyclic dependencies can be disallowed. Dependencies of standard exploits on DoS exploits can also be disallowed.

The fourth section of the master exploit list **250** can include the exploits that only consume resources. In step **1820**, the plug-in engine **275** can run these plug-in exploits **291** last.

Conclusion

The present invention has been described in relation to particular embodiments which are intended in all respects to be illustrative rather than restrictive.

Alternative embodiments will become apparent to those skilled in the art to which the present invention pertains without departing from its spirit and scope. Accordingly, the scope of the present invention is defined by the appended claims rather than the foregoing description.